Complexity Issues in Automated Addition of Time-Bounded Liveness Properties[1]

Borzoo Bonakdarpour and Sandeep S. Kulkarni

Software Engineering and Network Systems Laboratory,
Department of Computer Science and Engineering,
Michigan State University,
East Lansing, MI 48824, USA
Email: {borzoo, sandeep}@cse.msu.edu
http://www.cse.msu.edu/~{borzoo,sandeep}

Abstract. In this paper, we concentrate on synthesis of real-time programs modeled by Alur and Dill timed automata for automatic addition of different types of time-bounded liveness properties. Time-bounded liveness (also called time-bounded response) – that *something good* will happen soon, in a certain amount of time – captures a wide range of requirements for specifying real-time and embedded systems. We show that the problem of automatic addition of a time-bounded liveness property to a given timed automaton while maintaining maximal nondeterminism is NP-hard in the size of locations of the input automaton. Furthermore, we show that by relaxing the maximality requirement we can devise a sound and complete algorithm that adds a time-bounded liveness property to a given timed automaton, while preserving its existing MTL specification. This synthesis method is useful in adding properties that are later discovered as a crucial part of a program. Moreover, we show that addition of interval time-bounded liveness, where the good thing should not happen sooner than a certain amount of time, is also NP-hard in the size of locations even without maximal nondeterminism. Finally, we show that adding time-bounded and interval time-bounded as well as unbounded liveness properties are all PSPACE-complete in the size of the input timed automaton.

Keywords: Program transformation, Program synthesis, Timed automata, Real-time, Bounded liveness, Bounded response, Formal methods.

1 Introduction

Automated program synthesis is the problem of designing an algorithmic method to find a program that satisfies a required behavior. Such automated synthesis is desirable, as it ensures that the synthesized program is correct by construction. The synthesis problem has mainly been studied in two contexts: synthesizing programs from specification, where the entire specification is given, and synthesizing programs from existing programs along with a fully or partially available new specification. In approaches where the entire specification must be available, changes in specification, e.g., addition of a new property, requires us to begin from scratch. By contrast, in the latter approach, it is possible to *reuse* an existing program (and, hence, the previous efforts made for synthesizing the existing program). Since it may not be possible to anticipate all the necessary required properties at design time, this approach is especially useful in program maintenance, where the program needs to be modified to add a new property of interest.

In order to *add* a new property of interest to a program there are two ways: (1) *comprehensive redesign*, where the designer introduces new behaviors (e.g., by introducing new variables, or adding new computation paths), or (2) *local redesign*, where the designer removes behaviors that violate the property of interest, but does not add any new behaviors. While the former requires the

[1] This work was partially sponsored by NSF CAREER CCR-0092724, DARPA Grant OSURS01-C-1901, ONR Grant N00014-01-1-0744, NSF grant EIA-0130724, and a grant from Michigan State University.

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
2006		**00-00-2006 to 00-00-2006**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Complexity Issues in Automated Addition of Time-Bounded Liveness Properties	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Michigan State University,Department of Computer Science and Engineering,East Lansing,MI,48824**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**16**	

designer to verify all other properties of the new program, the latter ensures that certain existing properties (e.g., LTL and MTL) are preserved. Local redesign is especially applicable when the original program is designed manually, e.g., for ensuring that the original program is efficient. Moreover, with this approach, existing computations are preserved and, hence, it has the potential to preserve the efficiency of the original program.

Depending upon the choice of formulation of the problem and expressiveness of specifications and programs, the class of complexity of synthesis methods varies from polynomial time to undecidability. In this paper, we focus on complexity issues in synthesis methods that add properties typically used for specifying timing constraints in real-time programs using local redesign. Precisely, we identify the cases where the complexity of such addition is manageable. More specifically, we study the problem of incremental addition of *time-bounded liveness* properties (also called *time-bounded response*) – that something good will happen soon, in a certain amount of time – to Alur and Dill timed automata [1], while preserving their existing Metric Temporal Logic (MTL) specification [2]. This method will be especially desirable when an existing system is to be modified so that it meets new timing constraints (respectively, stronger timing constraints).

1.1 Related Work

In the context of untimed systems, in the pioneering work [3, 4], the authors propose methods for synthesizing the synchronization skeleton of programs from their temporal logic specification. More recently, in [5–7], the authors investigate algorithmic methods to locally redesign fault-tolerant programs using their existing fault-intolerant version and a partially available specification. In [8], the authors introduce a synthesis algorithm that adds UNITY properties [9] such as leads-to (which is an unbounded liveness property) to untimed programs.

Synthesis of real-time programs has mostly been formulated in the context of timed controller synthesis from game theoretical perspective. In the early works [10–12], the authors investigate the problem, where the given program (also called *plant*) is given by a *deterministic* timed automaton and the specification is modelled as a deterministic *internal* winning condition on the state space of the plant. The authors also assume that the controller can use *unlimited* resources (i.e., the number of new clocks and guards that compare the clocks to constants). Similarly, in [13], the authors solve the reachability problem in timed games. Deciding the existence of a winning condition with the formulation presented in [10–13] is shown to be EXPTIME-complete in [14].

In [15, 16], the authors address the problem of synthesizing timed controllers with limited resources. Similar to the aforementioned work, the plant is modelled by a deterministic timed automaton, but the specification is given by an *external* nondeterministic timed automaton that describes *undesired* behavior of the plant. With this formulation, the synthesis problem is 2EXPTIME-complete. However, if the given specification remains nondeterministic, but it describes *desired* behavior of the plant the problem turns out to be undecidable.

In [17], the authors propose a synthesis method for timed games, where the game is modelled as a timed automaton, the winning condition is described by TCTL-formulae, and unlimited resources are available. In [18], the authors consider concurrent two-person games given by a timed automaton played in real-time and provide symbolic algorithms for solving them with respect to all ω-regular winning conditions. In both approaches, deciding the existence of a winning strategy is EXPTIME-complete.

1.2 Contributions

The point of departure of our work from the above related work is as follows. In our work, we (i) consider the case where the entire specification of the program is not given to the synthesis algorithm; and (ii) model the notion of program by *nondeterministic* timed automata. In fact, we study how the *level of nondeterminism* affects the complexity of synthesis methods. High level of nondeterminism increases the potential of success in later manipulations such as adding another

time-bounded liveness property to the transformed program. Furthermore, unlike the related work, we model specifications in MTL. Moreover, the aim of our study is to identify the class of complexity of automated addition of different types of time-bounded liveness properties and possibly devising algorithms that can be used in tools for synthesizing real-time programs.

The main results in this paper are as follows:

- We show that adding a time-bounded liveness property while maintaining maximal nondeterminism is NP-hard in the size of locations of the given timed automaton.
- Based on the above result and the NP-hardness of adding two time-bounded liveness properties without maximal nondeterminism [2], we focus on addition of a single time-bounded liveness property to a time automaton without maximal nondeterminism. In fact, we present a surprising result that by dropping the maximality requirement we can devise a simple *sound* and *complete* transformation algorithm that adds a time-bounded liveness property to a timed automaton. The algorithm also ensures that the input timed automaton continues to satisfy its existing MTL properties. Since our algorithm is complete, if it fails to synthesize a program then it informs the designer a more comprehensive (and expensive) approach *must* be used. Moreover, since the complexity of our algorithm is comparable with that of model checking, the algorithm has the potential to provide timely insight to the designer about how the given program needs to be modified to meet the required time-bounded liveness property. Thus, in this paper, we extend the results presented in [8] to the context of real-time programs.
- We show that adding *interval time-bounded liveness*, where the good thing should not happen sooner than a certain amount of time, is also NP-hard in the size locations of the given timed automaton even without maximal nondeterminism.
- We show that the problems of adding time-bounded and interval time-bounded as well as *unbounded liveness* (also called *leads-to*) properties are all PSPACE-complete in the size of the input timed automaton.

Table 1 compares the complexity of our approach and other synthesis methods in the literature.

Adding Bounded Liveness (This paper)	Direct Synthesis from MTL [19]	Timed control synthesis [15, 16]	Timed games [10, 12, 13, 17, 18]
PSPACE-*complete*	EXPSPACE-*complete*	2EXPTIME-*complete*	EXPTIME-*complete*

Table 1. Complexity of different approaches for synthesizing real-time systems.

Organization of the paper. In Section 2, we present the preliminary concepts. In Section 3, we formally state the problem of addition of an MTL property to an existing real-time program. We describe the NP-hardness result for adding time-bounded liveness with maximal nondeterminism in Section 4. Then, in Section 5, we present a sound and complete algorithm for adding time-bounded liveness to timed automata without maximal nondeterminism. In Section 6, we present the complexity of addition of interval time-bounded liveness and unbounded liveness properties. In Section 7, we answer the potential questions raised about our approach. Finally, we make the concluding remarks and discuss future work in Section 8.

2 Preliminaries

In this section, we present the preliminary concepts and formal definitions of real-time programs and specifications. Real-time programs are modeled by Alur and Dill timed automata [1]. Specifications are modeled by Metric Temporal Logic (MTL) [2].

[2] In [8], it is shown that adding two unbounded liveness properties to an untimed program is NP-hard. The same proof can be easily extended to the problem of adding two time-bounded liveness properties to a timed automaton.

Let AP be a set of *atomic propositions*. A state is a subset of AP. A *timed state sequence* is an infinite sequence of pairs $(\sigma, \tau) = (\sigma_0, \tau_0), (\sigma_1, \tau_1)...$, where σ_i ($i \in \mathbb{N}$) is a state and $\tau_i \in \mathbb{R}_{\geq 0}$ satisfies the following constraints:

1. *Initialization*: $\tau_0 = 0$.
2. *Monotonicity*: $\tau_i \leq \tau_{i+1}$ for all $i \in \mathbb{N}$.
3. *Progress*: For all $t \in \mathbb{R}_{\geq 0}$, there exists j such that $\tau_j \geq t$.

2.1 Metric Temporal Logic

We briefly recap the syntax and semantics of *point-based* MTL. Linear Temporal Logic (LTL) specifies the *qualitative part* of a program. MTL introduces real time by constraining temporal operators, so that one can specify the *quantitative part* as well. For instance, the constrained eventually operator $\Diamond_{[1,3]}$ is interpreted as "eventually within 1 to 3 time units both inclusive".

Syntax. Formulae of MTL are inductively defined by the grammar: $\phi ::= p \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \mathcal{U}_I \phi_2$, where $p \in AP$ and $I \subseteq \mathbb{R}_{\geq 0}$ is an open, closed, half-open, bounded, or unbounded interval with endpoints in $\mathbb{Z}_{\geq 0}$. For simplicity, we use $\Diamond_I \phi$ and $\Box_I \phi$ instead of $true\mathcal{U}_I\phi$ and $\neg\Diamond_I\neg\phi$. We also use pseudo-arithmetic expressions to denote intervals. For instance, "≤ 4" means $[0, 4]$.

Semantics. For an MTL formula ϕ and a timed state sequence $(\sigma, \tau) = (\sigma_0, \tau_0), (\sigma_1, \tau_1)...$, the satisfaction relation $(\sigma_i, \tau_i) \models \phi$ is defined inductively as follows:

$(\sigma_i, \tau_i) \models p$ iff $\sigma_i \models p$ ($\sigma_i \models p$ iff $p \in \sigma_i$ and we say σ_i is a p-state);
$(\sigma_i, \tau_i) \models \neg\phi$ iff $(\sigma_i, \tau_i) \not\models \phi$;
$(\sigma_i, \tau_i) \models \phi_1 \wedge \phi_2$ iff $(\sigma_i, \tau_i) \models \phi_1 \wedge (\sigma_i, \tau_i) \models \phi_2$
$(\sigma_i, \tau_i) \models \phi_1 \mathcal{U}_I \phi_2$ iff there exists $j > i$ such that $\tau_j - \tau_i \in I$ and $(\sigma_{i'}, \tau_{i'}) \models \phi_1$ for all i', where $i \leq i' < j$, and $(\sigma_j, \tau_j) \models \phi_2$

A timed state sequence (σ, τ) satisfies the formula ϕ if $(\sigma_0, \tau_0) \models \phi$.

The formula ϕ defines a set Σ of timed state sequences that satisfy ϕ. We call this set a *specification* (or *property*). In this paper, we focus on a standard class of properties of real-time programs defined as follows. An *interval time-bounded liveness* (or *interval time-bounded response*) property is of the form $\mathcal{L}_I \equiv \Box(p \rightarrow \Diamond_{[\delta_1,\delta_2]} q)$, where $p, q \in AP$ and $\delta_1, \delta_2 \in \mathbb{Z}_{\geq 0}$; i.e., it is always the case that a p-state is followed by a q-state within δ_2, but not sooner than δ_1 time units. A special case of $\mathcal{L}_I$ is in which $\delta_1 = 0$ known as *time-bounded liveness* property and is of the form $\mathcal{L}_B \equiv \Box(p \rightarrow \Diamond_{\leq \delta} q)$; i.e., it is always the case that a p-state is followed by a q-state within δ time units. Furthermore, an *unbounded liveness* (or *leads-to*) property is defined as $\mathcal{L}_\infty \equiv \Box(p \rightarrow \Diamond_{[0,\infty)} q)$; i.e, it is always the case that a p-state is eventually followed by a q-state.

2.2 Timed Automata

For a set of clock variables X, the set $\Phi(X)$ of *clock constraints* φ is inductively defined by the grammar:

$$\varphi ::= x \leq c \mid x \geq c \mid x < c \mid x > c \mid \varphi \wedge \varphi$$

where $x \in X$ and $c \in \mathbb{Z}_{\geq 0}$. A *clock valuation* is a function $\nu : X \rightarrow \mathbb{R}_{\geq 0}$ that assigns a real value to each clock variable. Furthermore, for $\tau \in \mathbb{R}_{\geq 0}$, $\nu + \tau = \nu(x) + \tau$ for every clock x. Also, for $Y \subseteq X$, $\nu[Y := 0]$ denotes the clock valuation for X which assigns 0 to each $x \in Y$ and agrees with ν over the rest of the clock variables in X.

Definition 2.1. A *timed automaton* $\mathcal{A}$ is a tuple $\langle L, L^0, \psi, X, E \rangle$, where

- L is a finite set of *locations*,
- $L^0 \subseteq L$ is a set of initial locations,

- $\psi : L \rightarrow 2^{AP}$ is a labeling function assigning to each location the set of atomic propositions true in that location,
- X is a finite set of clocks, and
- $E \subseteq (L \times 2^X \times \Phi(X) \times L)$ is a set of *switches*. A switch $\langle s_0, \lambda, \varphi, s_1 \rangle$ represents a transition from location s_0 to location s_1 under clock constraint φ over X, such that it specifies when the switch is enabled. The set $\lambda \subseteq X$ gives the clocks to be reset with this switch. □

The semantics of a timed automaton is as follows. A *state* of a timed automaton is a pair (s, ν), such that s is a location and ν is a clock valuation for X at location s. The labeling function for states is defined by $\psi'((s, \nu)) = \psi(s)$. Thus, if $p \in \psi(s)$, s is a p-location (i.e., $s \models p$) and (s, ν) is a p-state for all ν. Since the domain of clock variables ranges over the real numbers, the *state space* of $\mathcal{A}$ is infinite. An initial state of $\mathcal{A}$ is (s_{init}, ν_{init}) where $s_{init} \in L^0$ and ν_{init} maps the value of all clocks in X to 0. *Transitions* of $\mathcal{A}$ are of the form $(s_0, \nu_0) \rightarrow (s_1, \nu_1)$. They are classified into two types:

- **Delay (elapse of time):** for a state (s, ν) and a time increment $\tau \in \mathbb{R}_{\geq 0}$, $(s, \nu) \xrightarrow{\tau} (s, \nu + \tau)$.
- **Location switch:** for a state (s_0, ν) and a switch $(s_0, \lambda, \varphi, s_1)$ such that ν satisfies the clock constraint φ, $(s_0, \nu) \rightarrow (s_1, \nu[\lambda := 0])$.

We use the well-known *railroad crossing problem* from the literature as a running demonstration throughout the paper. The original problem comprised of three timed automata, but we only consider the TRAIN automaton (cf. Figure 1-a). The TRAIN automaton models the behavior of a train approaching a railroad crossing. Initially, the train is far from the gateway of the crossing. It announces approaching the gateway by resetting the clock variable x. The train is required to start crossing the gateway after at least 2 minutes. It passes the gateway at least 3 minutes after approaching the gateway. Finally, there is no constraint on reaching the initial location.

We now define what it means for a timed automaton $\mathcal{A}$ to satisfy an MTL specification Σ. An infinite sequence $(s_0, \nu_0, \tau_0), (s_1, \nu_1, \tau_1)...$, where $\tau_i \in \mathbb{R}_{\geq 0}$, is a *computation* of $\mathcal{A}$ iff $\forall j > 0 : (s_{j-1}, \nu_{j-1}) \rightarrow (s_j, \nu_j)$ is a transition of $\mathcal{A}$ and the sequence $\tau_0 \tau_1 ...$ satisfies initialization, monotonicity, and progress. We write $\mathcal{A} \models \Sigma$ and say that timed automaton $\mathcal{A}$ *satisfies* specification Σ iff every computation of $\mathcal{A}$ that starts from an initial location is in Σ. Thus, $\mathcal{A} \models (\Box(p \rightarrow \Diamond_{\leq \delta} q))$ iff any computation of $\mathcal{A}$ that reaches a p-state, reaches a q-state within δ time units. If $\mathcal{A} \not\models \Sigma$, we say $\mathcal{A}$ *violates* Σ.

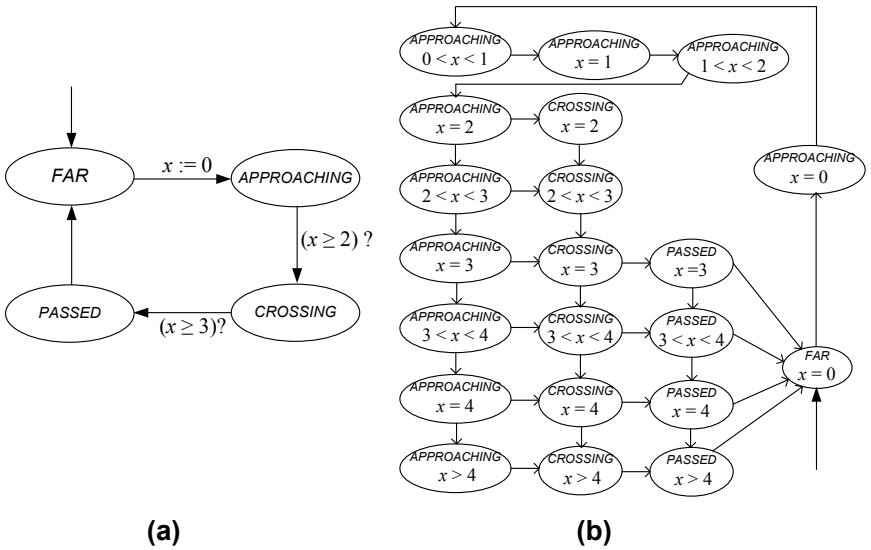

Fig. 1. (a) TRAIN automaton. (b) Region automaton of TRAIN automaton.

2.3 Region Automata

Given a timed automaton $\mathcal{A}\langle L, L^0, \psi, X, E\rangle$, to check whether a location s_1 is *reachable* from another location s_0, we must determine if there is a computation that starts from s_0 and reaches s_1 in the infinite state space. The solution to this reachability problem involves construction of a finite quotient proposed in [1]. This construction uses an equivalence relation, called *region equivalence* (denoted $\cong$), on the state space that equates two states with the same location, is defined over the set of all clock valuations for X. For two clock valuations ν and μ, $\nu \cong \mu$ iff:

1. $\forall x \in X : ((\lfloor \nu(x) \rfloor = \lfloor \mu(x) \rfloor) \ \vee \ (\nu(x), \mu(x) > c_x))$.
2. $\forall x, y \in X \ : \ ((\nu(x) < c_x \ \wedge \ \nu(y) < c_y)) : (\langle \nu(x) \rangle \ < \ \langle \nu(y) \rangle \ \text{iff} \ \langle \mu(x) \rangle \ < \ \langle \mu(y) \rangle)$.
3. $\forall x \in X : \nu(x) \ < \ c_x : (\langle \nu(x) \rangle = 0 \ \text{iff} \ \langle \mu(x) \rangle = 0)$.

where c_x is the largest integer c, such that x is compared with c in some clock constraint in $\mathcal{A}$, $\langle \tau \rangle$ denotes the fractional part, and $\lfloor \tau \rfloor$ denotes the integral part of τ and for any $\tau \in \mathbb{R}_{\geq 0}$. A *clock region* for $\mathcal{A}$ is an equivalence class of clock valuations induced by $\cong$. Note that, there are only finite number of regions. Also, region equivalence is a *time-abstract bisimulation* [1].

A *region* is a pair (s, ρ), where s is a location and ρ is a clock region. If s is a p-location, we say that (s, ρ) is a p-region. Using the region equivalence relation, we construct the *region automaton* of $\mathcal{A}$ (denoted $R(\mathcal{A})$) as follows. Vertices of $R(\mathcal{A})$ are regions. Edges of $R(\mathcal{A})$ are of the form $(s_0, \rho_0) \rightarrow (s_1, \rho_1)$ iff for some clock valuations $\nu_0 \in \rho_0$ and $\nu_1 \in \rho_1$, $(s_0, \nu_0) \rightarrow (s_1, \nu_1)$ is a transitions of $\mathcal{A}$. We note that, the size of a region automaton is in polynomial (respectively, exponential) order of its corresponding timed automaton, space-wise (respectively, time-wise). Figure 1-b shows the region automaton of the TRAIN automaton.

We say a region (s_0, ρ_0) of region automaton $R(\mathcal{A})$ is a *deadlock region* iff for all regions (s_1, ρ_1), there does not exist an edge of the form $(s_0, \rho_0) \rightarrow (s_1, \rho_1)$.

A clock region β is a *time-successor* of a clock region α iff for each $\nu \in \alpha$, there exists $\tau \in \mathbb{R}_{\geq 0}$, such that $\nu + \tau \in \beta$, and $\nu + \tau' \in \alpha \cup \beta$ for all $\tau' < \tau$. We call a region (s, ρ) a *boundary region*, if for each $\nu \in \rho$ and for any $\tau \in \mathbb{R}_{\geq 0}$, ν and $\nu + \tau$ are not equivalent. A region is *open*, if it is not a boundary region. A region (s, ρ) is called *end region*, if $\nu(x) > c_x$ for all clocks x. For instance, in Figure 1-b, $(APPROACHING, x = 2)$ is a boundary region, $(CROSSING, 3 < x < 4)$ is an open region, and $(PASSED, x > 4)$ is an end region.

3 Problem Statement

Given are a timed automaton $\mathcal{A}\langle L, L^0, \psi, X, E\rangle$ and an MTL property $\mathcal{L}$ (either $\mathcal{L}_I$, $\mathcal{L}_B$, or $\mathcal{L}_\infty$). Our goal is to find a timed automaton $\mathcal{A}'\langle L', L'^0, \psi', X', E'\rangle$, such that $\mathcal{A}' \models \mathcal{L}$ and for any MTL specification Σ, if $\mathcal{A} \models \Sigma$ then $\mathcal{A}' \models \Sigma$.

We now explain how we formulate the problem. Since we require that $\mathcal{A}' \models \Sigma$, if L' contains locations that are not in L, then $\mathcal{A}'$ may include computations that are not in Σ and as a result, $\mathcal{A}'$ may violate Σ. Hence, we require that $L' \subseteq L$ and $L'^0 \subseteq L^0$. Moreover, if E' contains switches that are present in E, but are guarded by weaker timing constraints, or E' contains switches that are not present in E at all then $\mathcal{A}'$ may include computations that are not in Σ. Hence, we require that E' contains a switch $\langle s_0, \lambda, \varphi', s_1\rangle$, if there exists $\langle s_0, \lambda, \varphi, s_1\rangle$ in E, such that φ' is stronger than φ. Furthermore, extending the state space of $\mathcal{A}$ by introducing new clock variables under the above circumstances is legitimate. Finally, we require ψ' to be equivalent to ψ. Thus, the synthesis problem is as follows:

Problem Statement 3.1. Given $\mathcal{A}\langle L, L^0, \psi, X, E\rangle$ and an MTL property $\mathcal{L}$, identify $\mathcal{A}'\langle L', L'^0, \psi', X', E'\rangle$ such that

$(C1)$ $L' \subseteq L$, $L'^0 \subseteq L^0$
$(C2)$ $\psi' = \psi$
$(C3)$ $X \subseteq X'$

$(C4)$ $\forall\langle s_0, \lambda, \varphi', s_1\rangle \mid \langle s_0, \lambda, \varphi', s_1\rangle \in E' :$
$(\exists\, \langle s_0, \lambda, \varphi, s_1\rangle \mid \langle s_0, \lambda, \varphi, s_1\rangle \in E : (\varphi' \Rightarrow \varphi))$
$(C5)$ $\mathcal{A}' \models \mathcal{L}$
$(C6)$ For any MTL specification Σ: $((\mathcal{A} \models \Sigma) \Rightarrow (\mathcal{A}' \models \Sigma))$ □

Note that, based on Problem Statement 3.1, since we allow synthesis methods to remove states and transitions of a timed automaton, such methods are appropriate to analyze linear types of temporal logic such as MTL. In fact, constraints of Problem Statement 3.1 do not suffice to reason about existential properties of a program expressed in branching-time temporal logics such as TCTL. We will discuss this issue in Section 7 (cf. second question) in detail.

Remark 3.2. The results in this paper can be easily extended to the case where all initial locations are preserved in the synthesized automaton. We discuss this issue in detail in proofs of theorems 5.1 and 5.2, and in Remark 5.3.

4 Adding Time-Bounded Liveness Properties with Maximal Nondeterminism

In this section, we show that the synthesis problem in Problem Statement 3.1 for adding a time-bounded liveness property while maintaining maximal nondeterminism is NP-hard in the size of locations of the input timed automaton. We show this result by a reduction from the Vertex Splitting Problem [20] in directed acyclic graphs (DAG).

Given a timed automaton $\mathcal{A}$ and property $\mathcal{L}_B \equiv \Box(p \rightarrow \Diamond_{\leq\delta} q)$, we say that the synthesized timed automaton $\mathcal{A}'$ is *maximally nondeterministic* iff $\mathcal{A}'$ meets all the constraints of Problem Statement 3.1 and its set of transitions is maximal. Maintaining maximal nondeterminism is desirable in the sense that it increases the potential for further successful manipulations of a synthesized program. Note that, although we defined maximality in terms of transitions of a timed automaton, one may define it in terms of reachable locations or behaviors of a timed automaton. We discuss this issue in Section 7 in detail.

The DAG Vertex Splitting Problem (DVSP). Let $G\langle V, A\rangle$ be a weighted DAG and v_s, v_t be arbitrary source and target vertices in G. Let G/Y denote the DAG when each vertex $v \in Y$ is split into vertices v^{in} and v^{out} such that all arcs $(v, u) \in A$, where $u \in V$, are replaced by arcs of the form (v^{out}, u) and all arcs $(w, v) \in A$, where $w \in V$, are replaced by arcs of the form (w, v^{in}). In other words, the outgoing arcs of v now leave vertex v^{out} while the incoming arcs of v now enter v^{in}, and there is no arc between v^{in} and v^{out}. The DAG vertex splitting problem is to find a vertex set Y, where $Y \subseteq V$ and $|Y| \leq i$ (for some positive integer i), such that the length of the longest path of G/Y from v_s to v_t is bounded by a prespecified value d. In [20], the authors show that DVSP is NP-hard.

We now show that the problem of adding of a time-bounded liveness property while maintaining maximal nondeterminism is NP-hard.

Instance. A timed automaton $\mathcal{A}\langle L, L^0, \psi, X, E\rangle$, a time-bounded liveness property $\mathcal{L}_B \equiv \Box(p \rightarrow \Diamond_{\leq\delta} q)$, and a positive integer k, where $|E| \geq k$.

Maximally Nondeterministic Time-bounded Liveness Addition Problem (MNTLAP). Does there exist a timed automaton $\mathcal{A}'\langle L', L'^0, \psi', X', E'\rangle$, such that $|E'| \geq k$ and $\mathcal{A}'$ meets the constraints of Problem Statement 3.1?

Theorem 4.1: MNTLAP is NP-hard in the size of locations of the input timed automaton.

Proof. We reduce DVSP to MNTLAP. The reduction maps a weighted DAG $G\langle V, A\rangle$ and integers d and i to a timed automaton $\mathcal{A}$ and integers δ and k, respectively.

Mapping. Let $G\langle V, A\rangle$ be any instance of DVSP whose longest path is to be bounded by d. Let $l(a)$ be the length of arc $a \in A$. We construct a timed automaton $\mathcal{A}$ as follows (cf. Figure 2). Each vertex $v \in V$ is mapped to a pair of locations v^{in} and v^{out} in $\mathcal{A}$. The set of initial locations of $\mathcal{A}$ is the singleton $L^0 = \{v_s^{in}\}$, where v_s is the source vertex in G. Switches of $\mathcal{A}$ consist of two types of switches as follows:

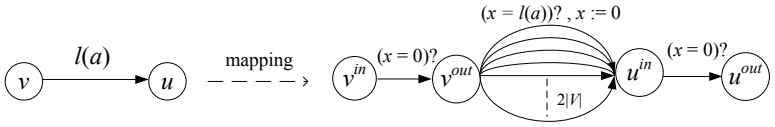

Fig. 2. Mapping DVSP to MNTLAP.

- We include switches of the form $v^{in} \xrightarrow{(x=0)?} v^{out}$ for all v in V. The clock constraint $(x = 0)$ is used to force computations of $\mathcal{A}$ not to wait at location v^{in}.
- We add $2|V|$ number of parallel switches of the form $v^{out} \xrightarrow{(x=l(a))?,\ x:=0} u^{in}$, for all arcs $a = (v, u) \in A$ of length $l(a)$.

Let the set of clock variables of $\mathcal{A}$ be the singleton $X = \{x\}$. Finally, let $v_s^{in} \models p$, $v_t^{out} \models q$, $k = i$, and $\delta = d$. Other locations may satisfy arbitrary atomic propositions except p and q.

Reduction. We need to show that vertex $v \in Y$ in G must be split if and only if the switch $v^{in} \xrightarrow{(x=0)?} v^{out}$ must be removed from $\mathcal{A}$. We distinguish two cases:

- *DVSP* $\longrightarrow$ *MNTLAP*: Suppose the answer to DVSP is the set Y, where $|Y| \leq i$. Hence, by splitting all $v \in Y$ the length of the longest path of G is at most d. Now, we show that we can synthesize a timed automaton $\mathcal{A}'$ from the mapped timed automaton $\mathcal{A}\langle L, \{v_s^{in}\}, \psi, \{x\}, E\rangle$ as an answer to MNTLAP. It is easy to see that if we remove switches of the form $v^{in} \xrightarrow{(x=0)?} v^{out}$ (for all $v \in Y$) from E to obtain E', the maximum delay between locations v_s^{in} and v_t^{out} in $\mathcal{A}'$ becomes at most δ. Recall that, $\delta = d$ and $i = k$. Therefore, $\mathcal{A}' \models \mathcal{L}_B$ and $|E'| \geq k$. Other constraints of Problem Statement 3.1 are immediately met by construction of $\mathcal{A}'$.
- *MNTLAP* $\longrightarrow$ *DVSP*: Suppose the answer to MNTLAP is $\mathcal{A}'\langle L', L'^0, \psi', \{x\}, E'\rangle$, where $|E'| \geq k$ and the maximum delay to reach v_t^{out} from v_s^{in} is at most δ. Note that, $L'^0 = \{v_s^{in}\}$, as v_s^{in} is the only initial location of $\mathcal{A}$. Since the number of switches removed from E is at most k and k is at most $|V|$, we could not have removed switches of the form $v^{out} \xrightarrow{(x=l(a))?,\ x:=0} u^{in}$. This is because there are $2|V|$ of such switches and their removal would not change the maximum delay. Hence, we should have removed switches of the form $v^{in} \xrightarrow{(x=0)?} v^{out}$ from E to bound the maximum delay. These switches actually identify the set Y of vertices that should be split in G; i.e, $Y = \{v \mid (v \in V) \wedge \langle v^{in}, v^{out}\rangle \in (E - E')\}$. It is easy to see that by removing the set Y from V the length of the longest path of G becomes at most d. □

5 Adding Time-Bounded Liveness Properties without Maximal Nondeterminism

In this section, we show that by relaxing the maximality constraint, we can solve the problem defined in Problem Statement 3.1 in polynomial time in the size of locations of the input timed automaton. More specifically, we present a sound and complete algorithm that adds a time-bounded liveness property to a given timed automaton, while preserving its existing MTL specification. Since our synthesis algorithm constructs and manipulates a specific weighted directed graph introduced by Courcoubetis and Yannakakis as a solution to the maximum delay problem in timed automata [21], we review this problem in Subsection 5.1. In Subsection 5.2, we describe our synthesis algorithm.

5.1 The Maximum Delay Problem in Timed Automata

The maximum delay problem is as follows. Given a timed automaton $\mathcal{A}$, a source location and clock valuation, what is the latest time that a target location can appear along a computation of

$\mathcal{A}$? For our purpose, we extend the proposed solution in [21] to the case where a set of source and target locations are given.

The algorithm in [21] works as follows. First, we construct the region automaton $R(\mathcal{A})\langle S, T\rangle$, where S is the set of regions and T is the set of edges. Then, we transform the region automaton to an ordinary weighted directed graph (called MaxDelay digraph). Let the subroutine ConstructMaxDelayGraph do this transformation as follows.

Construction of MaxDelay digraph. The subroutine ConstructMaxDelayGraph takes a region automaton $R(\mathcal{A})\langle S, T\rangle$, a set X of source regions, and a set Y of target regions, where $X, Y \subseteq S$, as input, and constructs a MaxDelay digraph $G\langle V, A\rangle$. Vertices of G consist of the regions in $R(\mathcal{A})$ with the addition of a source vertex v_s and a target vertex v_t.

Notation: We denote the weight of an arc (v_0, v_1) by $Weight(v_0, v_1)$. Let f denote a function that maps each region in $R(\mathcal{A})$ to its corresponding vertex in G; i.e., $f(r)$ is a vertex of G that represents region r in $R(\mathcal{A})$. Also, let f^{-1} denote the inverse of f; i.e., $f^{-1}(v)$ is the region of $R(\mathcal{A})$ that corresponds to vertex v in G. Likewise, let F be a function that maps a set of regions in $R(\mathcal{A})$ to the corresponding set of vertices in G and F^{-1} be its inverse. Finally, for a boundary region r with respect to clock variable x, we denote the value of x by $r.x$ (equal to some constant in $\mathbb{Z}_{\geq 0}$).

Arcs of G consist of the following:

- Arcs of weight 0 from v_s to all vertices in $F(X)$, and from all vertices in $F(Y)$ to v_t.
- Arcs of weight 0 from v_0 to v_1, if $f^{-1}(v_0) \rightarrow f^{-1}(v_1)$ is a location switch in $R(\mathcal{A})$.
- Arcs of weight $c' - c$, where $c, c' \in \mathbb{Z}_{\geq 0}$ and $c' > c$, from v_0 to v_1, if $f^{-1}(v_0)$ and $f^{-1}(v_1)$ are both boundary regions with respect to clock variable x_i, such that $f^{-1}(v_0).x_i = c$, $f^{-1}(v_1).x_i = c'$, and there is a path in $R(\mathcal{A})$ from $f^{-1}(v_0)$ to $f^{-1}(v_1)$, which does not reset x_i. It suffices to only consider the case where $c' - c = 1$.
- Arcs of weight $c' - c - \epsilon$, where $c, c' \in \mathbb{Z}_{\geq 0}$, $c' > c$, and $\epsilon \ll 1$, from v_0 to v_1 , if (1) $f^{-1}(v_0)$ is a boundary region with respect to clock variable x_i, (2) $f^{-1}(v_1)$ is an open region whose time-successor $f^{-1}(v_2)$ is a boundary region with respect to clock variable x_i, (3) $f^{-1}(v_0) \rightarrow f^{-1}(v_1)$ represents a delay transition in $R(\mathcal{A})$, and (4) $f^{-1}(v_0).x_i = c$ and $f^{-1}(v_2).x_i = c'$. Again, it suffices to only consider the case where $c' - c = 1$.
- Self-loop arcs of weight ∞ at vertex v, if $f^{-1}(v)$ is an end region.

In order to compute the maximum delay between X and Y, it suffices to find the longest distance between v_s and v_t in G. Note that, strongly connected components reachable from v_s containing an arc of nonzero weight cause maximum delay of infinity. As an example, Figure 3 shows the MaxDelay digraph the TRAIN automaton. The dotted arcs are a specific type of arcs and will be discussed in Subsection 5.2.

5.2 The Synthesis Algorithm

In this subsection, we present a sound and complete algorithm, Add_BoundedLiveness (cf. Figure 4), for solving the synthesis problem presented in Problem Statement 3.1 with respect to $\mathcal{L}_B \equiv \Box(p \rightarrow \Diamond_{\leq \delta} q)$. The core of the algorithm is straightforward. It begins with an empty digraph. Then, it invokes the subroutine ConstructSubgraph, which builds up a subgraph of the MaxDelay digraph by adding paths of length at most δ that start from the set of vertices that represents p-regions in G to the set of vertices that represents q-regions. Finally, it adds the rest of vertices and arcs while ensuring that no new paths from p-regions to q-regions are introduced. In order to ensure completeness, the algorithm preserves p-regions.

We now describe the algorithm in detail. First, in order to keep track of time whenever p becomes true, we add an extra clock variable t to $\mathcal{A}$ as a timer. Moreover, the maximum value that t would be compared with is δ (lines 1-3). Note that, since the length of a path in MaxDelay digraph is equal to the time elapsed along regions, our algorithm works correctly even if t is reset in between a p-state and a q-state (e.g., a computation that goes from a p-state to a $(\neg p)$-state, then

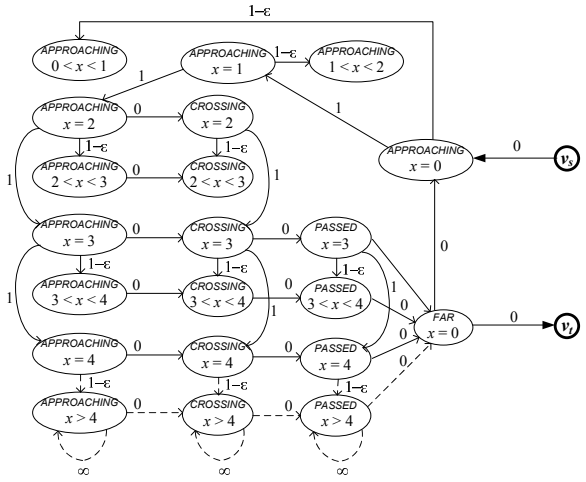

Fig. 3. MaxDelay digraph of TRAIN automaton.

again to a p-state, and finally to a q-state). Next, we construct the region automaton $R(\mathcal{A})\langle S, T\rangle$, where S is the set of regions and T is the set of edges (Line 4).

Notation: Since a location $s \in L$ may appear in a set of regions, in order to determine the source and target regions for computing maximum delay, we need to identify those regions where p and q become true. The function $g : AP \rightarrow 2^S$ calculates a set of such regions for an arbitrary atomic proposition ap as follows:

$$g(ap) = \{(s_1, \rho_1) \mid (s_1 \models ap) \ \wedge \\ (\exists\ (s_0, \rho_0) \mid (((s_0, \rho_0), (s_1, \rho_1)) \in T)\ :\ (s_0 \not\models ap))\}$$

We now reduce our problem to the problem of bounding the length of longest path in ordinary weighted digraphs. Towards this end, we first generate the MaxDelay digraph $G\langle V, A\rangle$ (Line 6), as described in Subsection 5.1. Then, we invoke the subroutine ConstructSubgraph (Line 7), where we construct a subgraph of G, which meets the required response time.

The subroutine ConstructSubgraph (lines 20-32) takes a MaxDelay digraph G and two integers δ and n as input. It generates a subgraph G' whose longest path from v_s to v_t is bounded by δ. Recall that v_s and v_t are additional source and target vertices connected to $F(g(p))$ and $F(g(q))$, respectively. Since enumerating all paths from v_s to v_t to test their lengths costs an exponential exhaustive search, we begin with an empty digraph and add a certain number of paths in polynomial order of $|S|$. To this end, first, we include the shortest path from each vertex in $F(g(p))$ to v_t, provided its length is at most δ (lines 21-24). In case there exists a vertex v in $F(g(p))$ from where there does not exist such a path to v_t, $f^{-1}(v)$ becomes a deadlock region.

In order to increase the level of nondeterminism, we now include additional n shortest paths whose length is at most δ. However, every time we add a path, we need to test that this path does not create new paths of length greater than δ or cycles containing an edge of nonzero weight (lines 25-29). One can interpret the integer n, as a level of nondeterminism; i.e., the more paths we add, the more nondeterminism we gain in the synthesized timed automaton. Next, we can safely add the rest of the vertices and arcs to G' (lines 30-32) while ensuring that no new paths are added from v_s to v_t.

After invoking ConstructSubgraph, we transform G' back to a region automaton $R(\mathcal{A}')$ (lines 8-10). Next, due to pruning some vertices and arcs in ConstructSubgraph, we remove deadlock regions from $R(\mathcal{A}')$ using a backward reachability analysis (lines 11, 12). However, in order to ensure that this removal does not break the completeness of our algorithm, we should consider the case where a q-region r_0 becomes a deadlock region. In this case, it is possible that all the regions along a path that starts from a region in $g(p)$ and ends at r_0 become deadlock

```
Add_BoundedLiveness(A⟨L, L^0, ψ, X, E⟩ : timed automata, n : integer, L_B ≡ □(p → ◇_{≤δ} q))
{
    X = X ∪ {t};                                                                                    (1)
    c_t := δ;                                                                                       (2)
    ∀⟨s_0, λ, φ, s_1⟩ | (⟨s_0, λ, φ, s_1⟩ ∈ E ∧ (s_0 ⊭ p ∧ s_1 ⊨ p)) : λ := λ ∪ {t};                (3)
    R(A)⟨S, T⟩ := ConstructRegionAutomaton(A);                                                      (4)
    Repeat
        IsQRemoved := false;                                                                        (5)
        G⟨V, A⟩ := ConstructMaxDelayGraph(R(A), g(p), g(q));      \\ Defined in Subsection 5.1      (6)
        G'⟨V', A'⟩ := ConstructSubgraph(G, δ, n);                                                   (7)

        R(A')⟨S', T'⟩ := {};                                                                        (8)
        S' := F^{-1}(V');                                                                           (9)
        T' := {(r_0, r_1) | (r_0, r_1) ∈ T ∧ (f(r_0), f(r_1)) ∈ A'} ∪
                  {(r_1, r_2) | (r_1, r_2) ∈ T ∧ (f(r_1), f(r_2)) ∉ A' ∧
                        ∃r_0 : Weight(f(r_0), f(r_1)) = 1 − ε};                                     (10)

        while (∃r_0 | r_0 ∈ S' : (∀r_1 | r_1 ∈ S' : (r_0, r_1) ∉ T'))                              (11)
            S' := S' − {r_0}; T' := T' − {(r, r_0), (r_0, r) | r ∈ S'};                              (12)
            if r_0 ∈ g(q) then                                                                      (13)
                IsQRemoved := true;                                                                 (14)
                S := S − {r_0}; T := T − {(r, r_0), (r_0, r) | r ∈ S}; break;                       (15)
    until (IsQRemoved = false);

    if {(s, ρ) | (s, ρ) ∈ S' ∧ s ∈ L^0 ∧ (∀x, ν | (ν ∈ ρ ∧ x ∈ X) : ν(x) = 0)} = {} then          (16)
        declare failure; exit;                                                                      (17)
    A' := ConstructTimedAutomata(R(A'));                                                            (18)
    return A';                                                                                      (19)
}
ConstructSubgraph(G⟨V, A⟩ : MaxDelay digraph, δ, n : integer)
{
    G'⟨V', A'⟩ = {};                                                                                (20)
    for all vertices v such that (v_s, v) ∈ A                                                       (21)
        if the length the shortest path P from v to v_t is at most δ then                           (22)
            V' := V' ∪ {u | u is on P};                                                             (23)
            A' := A' ∪ {a | a is on P};                                                             (24)
    for k = 1 to n                                                                                  (25)
        if adding the kth shortest path does not create other paths of length
         greater than δ or cycles containing an edge of nonzero weight then                         (26)
            P := the kth shortest path of G from v_s to v_t;                                        (27)
            V' := V' ∪ {u | u is on P};                                                             (28)
            A' := A' ∪ {a | a is on P};                                                             (29)
    A' := A' ∪ {(u, v) | (u, v) ∈ A ∧ (u ∉ V' ∨ (u, v_t) ∈ A')};                                     (30)
    V' := (V' ∪ {u | (∃v : (u, v) ∈ A' ∨ (v, u) ∈ A')}) − {v_s, v_t};                                (31)
    return G'⟨V', A'⟩;                                                                              (32)
}
```

Fig. 4. The synthesis algorithm for adding time-bounded liveness.

regions. Thus, we need to find another path from that region in $g(p)$ to a region in $g(q)$ other than r_0. Hence, we remove r_0 from the set of regions of the original region automaton $R(\mathcal{A})$ and start over (lines 14, 15). In case the removal of deadlock regions leaves no initial regions, the algorithm declares failure and terminates (lines 16, 17). Otherwise, it constructs the timed automaton $\mathcal{A}'$ out of $R(\mathcal{A}')$ (lines 18, 19) and terminates successfully.

As a demonstration, let us consider the TRAIN automaton presented in Section 2 (cf. Figure 1-a). Our goal is to bound the delay of revisiting the initial location by at most 4 minutes. To this end, we add the property $\mathcal{L}_B \equiv \Box(APPROACHING \rightarrow \Diamond_{\leq 4} FAR)$ to the TRAIN automaton. Since the automaton already contains a clock that gets reset upon entering the location *APPROACHING*, we do not add an extra clock. However, we should have $c_x = 4$ when generating the region automaton (cf. Figure 1-b). Next, we construct the MaxDelay digraph (cf. Figure 3). In Figure 3, the dotted arcs contribute in violating the required response time. On the other hand, the solid arcs do not violate $\mathcal{L}_B$. It is easy to observe that by choosing $n = 12$, ConstructSubgraph includes all computations that satisfy $\mathcal{L}_B$. The rest of the procedure is constructing the new region automaton (cf. Figure 5-a) and then the final timed automaton (cf. Figure 5-b), which is straightforward .

Theorem 5.1: The algorithm Add_BoundedLiveness is sound.

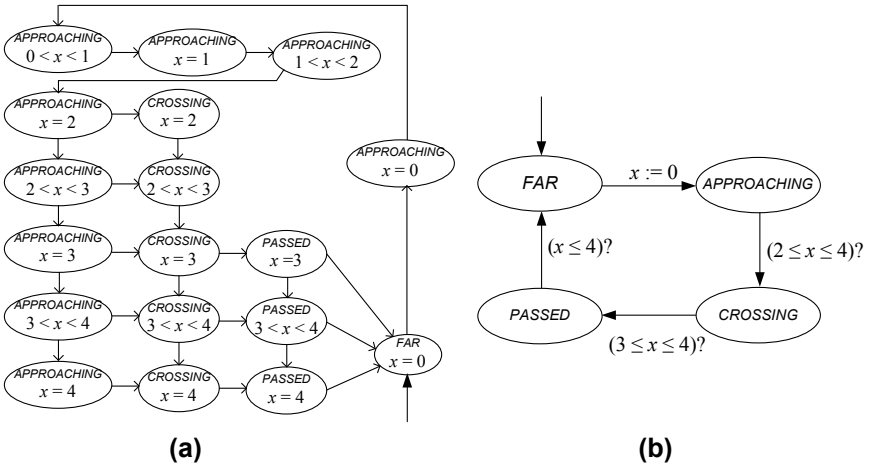

Fig. 5. (a) Synthesized region automaton (b) Synthesized TRAIN automaton.

Proof. We show that the timed automaton synthesized by Add_BoundedLiveness meets the constraints of Problem Statement 3.1:

- *Constraints* $C1...C3$: It is easy to observe that the algorithm Add_BoundedLiveness only removes locations of $\mathcal{A}$. Hence, $L' \subseteq L$ and $L'^0 \subseteq L^0$. Note that, pruning regions only change the guards of the associated switches and it does not affect reconstruction of $\mathcal{A}'$ such that $L' \subseteq L$. Also, we add an extra clock variable t. Hence, $X \subseteq X'$. Furthermore, the algorithm does not touch the labels of locations and, hence, $\psi' = \psi$.
- *Constraint* $C4$: The subroutine ConstructSubgraph may only remove regions or edges from a region automaton. This removal either removes a switch from the original timed automaton completely or makes some regions unreachable, which in turn strengthens the guard of one or more switches. Hence, the set of switches of $\mathcal{A}'$ meets the constraint $C4$.
- *Constraint* $C5$: The subroutine ConstructSubgraph ensures that the maximum delay of any computation that starts from a region in $g(p)$ and reaches a region in $g(q)$ is finite and bounded by the required response time in $\mathcal{L}_B$. Hence, we are assured that the synthesized timed automaton satisfies $\mathcal{L}_B$.
- *Constraint* $C6$: The algorithm removes deadlock regions from $R(\mathcal{A}')$. In other words, it ensures all computations of $\mathcal{A}'$ are infinite. Moreover, from constraints $C1...C4$, it follows that the algorithm does not introduce new computations to $\mathcal{A}'$. Thus, the set of computations of $\mathcal{A}'$ is a subset of the set of computations of $\mathcal{A}$ and, hence, for all MTL specifications Σ, if $\mathcal{A} \models \Sigma$ then $\mathcal{A}' \models \Sigma$ as well. □

Theorem 5.2: The algorithm Add_BoundedLiveness is complete.

Proof. In order to prove the completeness, we show that any initial location removed from the synthesized automaton must be removed. Observe that when a p-region is removed, there is no path from that region to a q-region where the delay is at most δ. It follows that such a region must be removed in any timed automaton that satisfies the constraints of Problem Statement 3.1. Furthermore, if removal of such a region causes another region to become a deadlock region then that region must be removed for satisfying the constraint $C5$. Continuing thus, if an initial region becomes a deadlocked region then it *must* be removed. Our algorithm declares failure when all initial locations are removed. Based on the above discussion, in this case, any timed automaton that satisfies the constraints of Problem Statement 3.1 cannot contain any of the initial locations from L^0. Since this is a contradiction, it follows that when Add_BoundedLiveness declares

failure, no solution exists for the given instance. Therefore, Add_BoundedLiveness is complete. □

Remark 5.3. If we were to preserve all initial locations (cf. Remark 3.2) then the algorithm is modified in this fashion where all initial locations are preserved and the remaining constraints from Problem Statement 3.1 are satisfied. However, the core of the above proofs still hold. To show soundness, in addition to those constraints, we need to check wether all initial locations are present. The completeness proof remains unchanged.

Theorem 5.4: The algorithm Add_BoundedLiveness is in P in the size of region automata.

Proof. The core of the algorithm is reachability analysis for a timed automaton. Deciding reachability of a location in timed automata is in P in the size of the region automaton [21]. Moreover, our synthesis algorithm involves finding shortest paths and the k shortest paths in an ordinary weighted digraph. Eppstein [22] proposes an algorithm that finds the k shortest paths (allowing cycles) in time $O(m + n \log n + k)$, where n is the number of vertices and m is the number of arcs of a given digraph. Note that, we require that k must be in polynomial order of the number of locations of the input timed automaton. Hence, one can implement a synthesis algorithm which runs in polynomial time in the qualitative part (locations), and polynomial space in the quantitative part of the input (timing constraints). □

Corollary 5.5: The problem of adding a time-bounded liveness property to a timed automaton is in PSPACE in the size of the input timed automaton. □

6 Adding Interval Time-Bounded and Unbounded Liveness Properties

We first consider automatic addition of an interval time-bounded liveness property $\mathcal{L}_I \equiv \Box(p \rightarrow \Diamond_{[\delta_1,\delta_2]} q)$ to a timed automaton, where $\delta_1 > 0$. As an intuition, let us use the algorithm Add_BoundedLiveness to add $\mathcal{L}_I$. Since the required response time has a lower bound, the subroutine ConstructSubgraph has to enumerate and ignore all the paths whose lengths are less than δ_1. Since there may exist many of these paths, this enumeration cannot be done in polynomial time in the size of region automata.

Theorem 6.1: The problem of adding an interval time-bounded liveness property to a timed automaton is NP-hard in the size of locations of the input timed automaton.

Proof. The proof is a simple reduction from the *longest path problem* [23] to an instance of the problem, where $\mathcal{L}_I \equiv \Box(p \rightarrow \Diamond_{[\delta_1,\infty)} q)$. Figure 6 illustrates the mapping of a digraph G to a timed automaton $\mathcal{A}$. It is easy to see that if G has a path of length at least δ_1 from a source vertex v_s to a target vertex v_t then $\mathcal{A}$ can be transformed to a timed automaton $\mathcal{A}'$ whose delay from v_s to v_t is at least δ_1 time units and vice versa. □

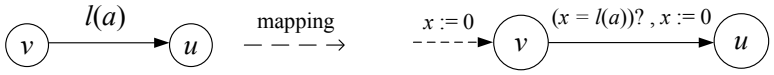

Fig. 6. Mapping the longest path problem to addition of interval time-bounded liveness.

Next, we discuss the problem of addition of unbounded liveness (also called leads-to) properties.

Theorem 6.2: The problem of addition of an unbounded liveness property to a timed automaton is PSPACE-complete in the size of the input timed automaton.

Proof. Since this problem is an instance of adding time-bounded liveness, membership to PSPACE follows from Corollary 5.5 immediately. We now show that the problem is PSPACE-hard. To this end, we reduce the *reachability problem* in timed automata [21] to an instance of our problem. In the reachability problem, our goal is to check whether a location s_1 is reachable from another location s_0 in a given timed automaton.

Mapping. Let the timed automaton $\mathcal{A}$ be any instance of the reachability problem. We map $\mathcal{A}$ to an instance of our problem as follows. Let $\mathcal{A}^*$ be an automaton identical to $\mathcal{A}$ with the following

modifications. Let $s_0 \models p$ and $s_1 \models q$. Other locations of $\mathcal{A}^*$ may satisfy arbitrary atomic propositions except p and q. Let s_0 be the only initial location of $\mathcal{A}^*$. We also add a self-loop at s_1.
Reduction. If s_1 is reachable from s_0 in $\mathcal{A}$ then there exists a computation in $\mathcal{A}^*$ that starts from s_0 and ends at s_1. A timed automaton $\mathcal{A}'$ constructed from this computation plus the self-loop at s_1 satisfies $\mathcal{L}_\infty$ and meets the constraints of Problem Statements 3.1. Now, we show the other direction. Let us assume that the answer to the decision problem is affirmative and we can synthesize a timed automaton $\mathcal{A}'$ from $\mathcal{A}^*$ such that $\mathcal{A}' \models \mathcal{L}_\infty$. Then $\mathcal{A}'$ should contain both s_0 and s_1. This means that s_1 is reachable from s_0. Otherwise, $\mathcal{A}'$ would not satisfy $\mathcal{L}_\infty$. □

Since an unbounded liveness property is an instance of time-bounded and interval time-bounded liveness properties, problems of adding those properties are also PSPACE-hard. This result is also valid about addition problems with maximal nondeterminism constraint, as synthesizing a timed automaton with at least one edge is an instance of the problem with maximal nondeterminism.
Corollary 6.3: The problems of adding time-bounded and interval time-bounded as well as unbounded liveness properties to a timed automaton with or without maximal nondeterminism are all PSPACE-complete in the size of the input timed automaton. □
Remark 6.4. The time complexity of adding an unbounded liveness property to a timed automaton with maximal nondeterminism in terms of transitions remains open in this paper. However, we refer the reader to [8], where the authors introduce a synthesis algorithm for adding leads-to properties to an untimed program, while maintaining maximal nondeterminism in terms of states of the given program.

We summarize the complexity of problems of addition of different types of liveness properties in Table 2.

7 Discussion

In this section, we address some of the questions raised about the formulation of the problem and the synthesis method presented in Section 5.

1- *How does our work fit in the context of related work?*

Our formulation of the problem (cf. Section 3) is different from those in [10–12, 15–17]. Intuitively, we manipulate a timed automaton inside its state space, so that it satisfies a newly desired property. By contrast, in [15, 16], the goal is synthesizing a timed controller (which is a timed automaton itself), such that its synchronized product with the plant satisfies a given specification. Hence, this formulation requires both plant and controller to be *deterministic* timed automata, whereas in our model, we synthesize *nondeterministic* timed automata even with a specified level of nondeterminism. Furthermore, in [10–12], the winning condition is given on the state space of the plant, whereas in our approach, the new property is an external MTL formula. Moreover, in this paper, our goal is to study the complexity issues and develop algorithms for adding various types of a specific class of MTL properties that (we believe) can capture a wide range of requirements for specifying real-time programs. In fact, the complexity of our algorithm is less than those in [10–12, 15–17] (cf. tables 1, 2). Of course, this is achieved at the cost of expressiveness of specifications.

2- *After removing a subset of computations, how can we claim that the synthesized timed automaton continues to satisfy its old specification?*

This is because we consider a linear type of temporal logic. As mentioned in Section 2, an MTL formula Σ defines a set of timed state sequences. Note that, an automaton $\mathcal{A}$ satisfies specification Σ iff all computations of $\mathcal{A}$ are in Σ. Hence, a subset of computations of $\mathcal{A}$ satisfies Σ as well. In the context of the algorithm Add_BoundedLiveness, although it excludes some of the computations, since it ensures that all computations are infinite (by removing deadlock regions), it continues to satisfy its old MTL specification. A possible confusion is that "the given program (before synthesis) does not satisfy the time-bounded liveness property $\mathcal{L}$, but it does satisfy $\mathcal{L}$

after synthesis". Note, however, that "a program does not satisfy $\mathcal{L}$" cannot be expressed as "the program satisfies $\mathcal{L}'$", where $\mathcal{L}'$ is an MTL property. Also, if a given program satisfies $\neg\mathcal{L}$ then no computation of the program satisfies $\mathcal{L}$ and, hence, it is not possible to synthesize a program that satisfies $\mathcal{L}$. In such a case, the algorithm Add_BoundedLiveness declares failure. The same problem cannot be defined by branching-time temporal logics (e.g., TCTL), as "a program does not satisfy $\mathcal{L}$" can be expressed as "the program satisfies $\mathcal{L}'$", where $\mathcal{L}'$ is a TCTL property.

3- *In Section 4, we defined maximality in terms of reachable transitions. What are the other alternatives to model nondeterminism?*

It is also possible to define maximal nondeterminism in terms of reachable locations or behaviors. However, various definitions does not change the NP-harness result. In fact, many of the edge and vertex deletion problems are known to be NP-hard [20, 24, 25]. In particular, in case of maximal reachable locations, one can easily reduce the *vertex deletion problem* [20] to our synthesis decision problem. Moreover, in case of maximal number of behaviors, one can develop a reduction from the kth *shortest path problem* [23].

4- *How can we improve the state space explosion problem in our algorithm?*

Generation of detailed region automaton is usually not efficient. Zone automata [26] is a more efficient finite representation of timed automata used in model checking techniques. Since our goal was to evaluate complexity classes for adding time-bounded liveness, we focused on region automata. However, an interesting improvement step is modifying Add_BoundedLiveness, so that it manipulates a zone automaton rather than a detailed region automaton.

Time-Bounded Liveness		Unbounded Liveness		Interval Time-Bounded Liveness
Maximal (Sec. 4)	**NonMaximal** (Sec. 5)	**Maximal** (Sec. 6)	**NonMaximal** (Sec. 6)	(Sec. 6)
NP-hard	*P*	see Rem. 6.4	*P*	*NP-hard*

Table 2. Complexity of adding liveness properties in the size of region automata.

8 Conclusion and Future Work

In this paper, we focused on the problem of automatic addition of different types of time-bounded liveness properties (also called bounded response) to a timed automaton, while preserving its existing Metric Temporal Logic (MTL) specification. Unlike specification-based methods, in our approach, we start with an existing program rather than specification and, hence, the previous efforts made for synthesizing the input program are reused.

First, we showed the problem of addition of a time-bounded liveness property to a timed automaton while maintaining maximal nondeterminism is NP-hard in the size of locations of the input automaton. Then, we presented a simple sound and complete transformation algorithm that adds a time-bounded liveness property to a timed automaton (without maximal nondeterminism), such that the automaton continues to satisfy its existing MTL specification. The complexity of the algorithm is polynomial in the size of locations of the input timed automaton. Furthermore, we showed that the problem of addition of interval time-bounded liveness properties is also NP-hard. Moreover, we showed that adding time-bounded and interval time-bounded as well as unbounded liveness properties are all PSPACE-complete in the size of the input timed automaton.

In many hard real-time systems (e.g., mission-critical systems) meeting deadlines in the presence of faults is a necessity. As future work, we plan to study the problem of automatic addition of fault-tolerance to existing fault-intolerant real-time programs. More specifically, we plan to extend the theory of automated addition of fault-tolerance to untimed programs [5–7] to the context

of real-time programs. In particular, we will study how bounded-time recovery can be achieved in the presence of faults using the results presented in this paper.

Acknowledgment. The authors would like to thank Edith Elkind at Princeton University for her ideas on the NP-hardness result in Section 4.

References

1. R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
2. R. Alur and T.A. Henzinger. Real-Time Logics: Complexity and Expressiveness. *Information and Computation*, 10(1):35–77, May 1993.
3. E.A. Emerson and E.M. Clarke. Using branching time temporal logic to synthesis synchronization skeletons. *Science of Computer Programming*, 2(3):241–266, 1982.
4. Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 6(1):68–93, 1984.
5. S. S. Kulkarni and A. Arora. Automating the addition of fault-tolerance. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1926 of *LNCS*, pages 82–93, 2000.
6. S. S. Kulkarni, A. Arora, and A. Chippada. Polynomial time synthesis of Byzantine agreement. In *20th Symposium on Reliable Distributed Systems*, pages 130–140, 2001.
7. S. S. Kulkarni and A. Ebnenasir. Automated synthesis of multitolerance. In *International Conference on Dependable Systems and Networks*, pages 209–219, 2004.
8. A. Ebnenasir, S. S. Kulkarni, and B. Bonakdarpour. Revising UNITY programs: Possibilities and limitations. In *9th International Conference on Principles of Distributed Systems*, 2005. To appear.
9. K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.
10. H. Wong-Toi and G. Hoffmann. The control of dense real-time discrete event systems. In *30th Conf. Decision and Control*, pages 1527–1528, Brighton, UK, 1991.
11. O. Maler, A. Pnueli, and J. Sifakis. On the synthesis of discrete controllers for timed systems. *12th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 229–242, 1995.
12. E. Asarin, O. Maler, A. Pnueli, and J. Sifakis. Controller synthesis for timed automata. *IFAC Symposium on System Structure and Control*, pages 469–474, 1998.
13. E. Asarin and O. Maler. As soon as possible: Time optimal control for timed automata. In *Hybrid Systems: Computation and Control*, volume 1569 of *LNCS*, pages 19–30, 1999.
14. T. A. Henzinger and P. W. Kopke. Discrete-time control for rectangular hybrid automata. *Theoretical Computer Science*, 221(1-2):369–392, 1999.
15. D. D'Souza and P. Madhusudan. Timed control synthesis for external specifications. In *19th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 571–582, 2002.
16. P. Bouyer, D. D'Souza, P. Madhusudan, and A. Petit. Timed control with partial observability. In *Computer Aided Verification (CAV)*, pages 180–192, 2003.
17. M. Faella, S. LaTorre, and A. Murano. Dense real-time games. In *Logic in Computer Science (LICS)*, pages 167–176, 2002.
18. L. de Alfaro, M. Faella, T. A. Henzinger, R. Majumdar, and M. Stoelinga. The element of surprise in timed games. In *14th International Conference on Concurrency Theory (CONCUR)*, 2003.
19. R. Alur, T. Feder, and T.A. Henzinger. The benefits of relaxing punctuality. *Journal of the ACM*, 43(1):116–146, 1996.
20. D. Paik, S.M. Reddy, and S. Sahni. Deleting vertices to bound path length. *IEEE Transation on Computers*, 43(9):1091–1096, 1994.
21. C. Courcoubetis and M. Yannakakis. Minimum and maximum delay problems in real-time systems. *Computer-Aided Verificaion*, LNCS 575:399–409, 1991.
22. D. Eppstein. Finding the k shortest paths. *SIAM Journal of Computing*, 28(2):652–673, 1999.
23. M.R. Garey and D.S. Johnson. *Computers and Intractability:A Guide to the Theory of NP-Completeness*. W. H. Freeman, New York, 1979.
24. M. Yannakakis. Node- and edge-deletion NP-complete problems. In *Conference Record of the Tenth Annual ACM Symposium on Theory of Computing*, pages 253–264. ACM press, 1978.
25. A. Natanzon, R. Shamir, and R. Sharan. Complexity classification of some edge modification problems. *Discrete Applied Mathematics*, 113(1):109–128, 2001.
26. R. Alur, C. Courcoubetis, N. Halbwachs, D. L. Dill, and H. Wong-Toi. Minimization of timed transition systems. In *International Conference on Concurrency Theory (CONCUR)*, pages 340–354, 1992.